

PARK User Manual

PARK is a pluggable framework for distributed computing. It is implemented in pure python, provides a simple and easily managed distributed environment for computational- intensive computing. The architecture of current PARK implementation is shown as in Fig. 1.

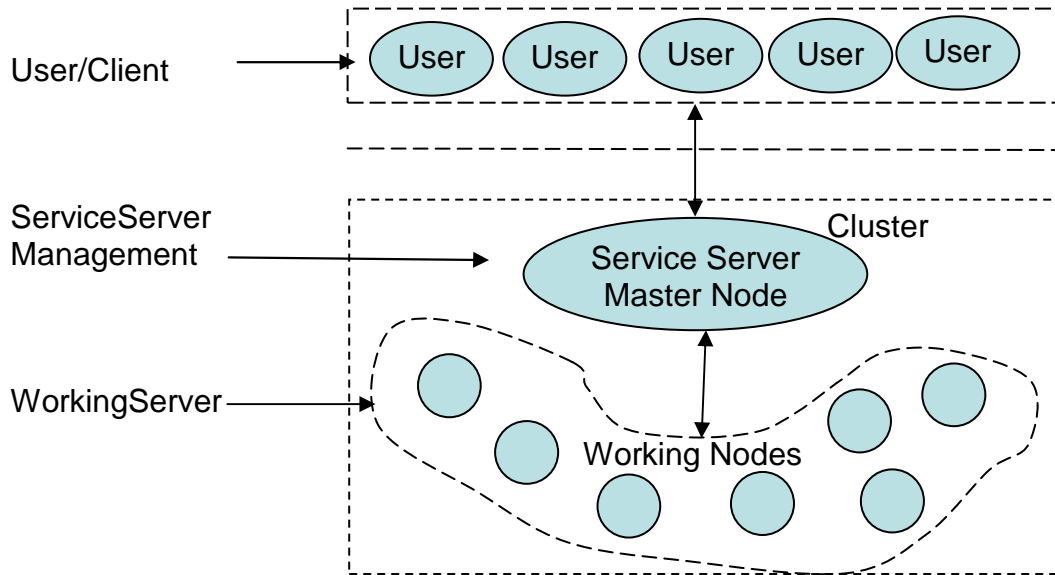


Fig. 1 The architecture of current PARK implementation

PARK is implemented in the standard client/server style and map/reduce model [further information, see “MapReduce: Simplified Data Processing on Large Clusters” at <http://labs.google.com/papers/mapreduce.html>]. The main part of PARK is the service management server, which can provide the job management and cluster management for the distributed computing. The service management server starts the working server on each working node in the cluster, manage the request job queue and the load-balance.

Server installation

The PARK server includes the service management server and the service working server. The service management server will run in the master node which can be connected to the outside client. It will start and stop the service working server, manage the job and cluster. The service working server runs on each working node within the cluster, provides the services to the service management server.

Pre-requisite:

python > 2.40 (For all cases)
Most services may need numpy / scipy.

Unix/Linux/Mac OS:

1. Download *park-x.y.z.tar.gz* or *park-x.y.z.tar.bz2* or *park-x.y.z.src.zip* from <http://chemnuc-20.umd.edu/~DANSE/download/index.html>

2. Unzip the file:

```
tar -xvzf park-x.y.z.tar.gz
```

3. Make the installation:

```
cd park-x.y.z  
make install
```

or

```
setup.py install --install-purelib=home_directory_of_park
```

The command *make install* is equivalent to *setup.py install --install-purelib=~*
It will install park in directory *~/park*.

4. Delete the build files (optional):

```
cd ../  
rm -r park-x.y.z  
rm park-x.y.z.tar.gz
```

5. Setup environments to run park(optional): We assume that park is installed in *~/park*.

```
cd park/config
```

Edit *hosts* file, so only the available nodes are included as the working nodes.

There are several other examples that can be use where the only the local host is used as the working nodes, whose number of cpus may be 1, 2, 4, 8, 16 (files *hosts.local*, *hosts.local2*, *hosts.local4*, *hosts.local8*, *hosts.local16*).

6. Start the server:

```
cd park  
python parkServer.py
```

or

```
./parkServer.py
```

When there is no error, the service server is ready to accept the service request. The user can connect to the server and submit the service request. Make sure that the services are executable files. For example, in Unix system, use

```
chmod u+x *.py  
to make the services executable.
```

7. Testing (optional)

Using the echo client to testing the connection.

```
cd park/parkUI  
python echoClient.py
```

or

```
./echoClient.py
```

This will echo the contents of the request, which will be sent to the service applications as a string in the real application. The application will parse the input string, do the real work, and then send its results to the client.

8. Shut down the service server by **Ctrl-C** or **kill** command. Please use **kill** without **-9** command, which will also stop the working server program. Otherwise the working server will continue to work even the service server is killed.

Error shooting

- 1) Make sure that python and its environments are set correctly.
- 2) For cluster with multiple working nodes, make sure that RSH defined in `park/servers/envron.py` is set to the remote shell command. The default is `'rsh'`, and it may be changed to other remote shell command, such as `'ssh'`. Make sure that this remote shell command can start the remote command without the password.
- 3) [Errno 2] No such file or directory: `'~/park/config/hosts'`: no configure file hosts.
- 4) ERROR (111, 'Connection refused') or (socket.error:(111, 'Connection refused'))
 - a. The working server doesn't start.
 - b. netstat make sure that the port is not used.
- 5) ERROR (98, 'Address already in use')
 - a. The address is used.
 - b. The server cannot be started immediately after it is shutdown.
- 6) : No such file or directory:
 - a) Make sure the file is in the proper directory
 - b) Using `servers/switchFD.py` to change the `\r\n` string used in windows to `\r` in Unix

Windows OS:

The procedure to install PARK python source code is the same as in Unix under cygwin:

```
unzip park-x.y.z_src.zip  
setup.py install
```

It will install PARK in site-packages/park.

The executable file is also available for Windows platform (`parkServer.exe`).

User Manual for GUI Client

see Simple Instructions for more details, especially for Windows

Pre-requisite:

```
python > 2.40
wxPython > 2.8
matplotlib : svn version
```

1. Enter ~/park
2. Run the client application:

```
$python parkClient.py
```

A GUI program will popup.

For Windows executable files, the client application is:
`parkClient.exe`

parkWin.* is a program that runs on a single node in Windows platform. It first starts the PARK server, and then starts the PARK client.

The reflectometry and boxmin modules are needed to run parkClient.py. Extract these two models from

```
svn co svn://svn@danse.us/boxmin
svn co svn://svn@danse.us/reflectometry
```

Then go to the ~/reflectometry, run

```
./configure
make
```

This will create a python/ directory under home directory. Copy the ~/python/boxmin and ~/python/reflectometry to ~/park.

Sometimes the backend may need to be changed. Edit ~/park/client_auI/model/refl/mixbackend.py to select the available backend in matplotlib, which could be found at lib/python2.5/site-packages/matplotlib/backends.

All the source code of PARK can be accessed via:

```
svn co svn://svn@danse.us/park
```

Directory Structure

~/park Home directory for PARK
~/park/servers Directory for PARK servers
~/park/services Directory for PARK services
~/park/parkUI Directory for client-side script-based UI application
~/park/client_auI Directory for client-side wx/matplotlib based GUI application (version 0.3.*)
~/park/parkAui Directory for client-side wx/matplotlib based GUI application (version 0.4 and above)
~/park/examples Some examples

Example to develop applications on PARK:

- 1) Develop the client-side applications to submit the request, and represent the results.
- 2) Develop the server-side applications to do the real work.

1) Client-Side Application

The client-side application needs to build the job request, submit the request, then parse and represent the results. The *echoClient.py* in *parkUI* is such a simple client-side application, which will use the echo service to testing the network connection and environment setting:

```
#!/usr/bin/env python

#####
import time
import traceback
import socket
#####

from replyEcho import ParkReplyEcho
from networkThread import NetworkThread

#####
## default PARK servers:
DEFAULT_SERVERS = [socket.gethostname(),
                   'localhost', 'compufans.ncnr.nist.gov']
## default port number used in park
DEFAULT_PORT = 5400

## DEFAULT WAITING TIME
WAITING_TIME = 150
#####

#####
def main():
    server = (DEFAULT_SERVERS[0], DEFAULT_PORT)
    reply = ParkReplyEcho()

    networkThread = None

    try:
(1)    networkThread = NetworkThread(server, reply )
[2.1]    networkThread.start()
        print '\n --- Start network Thread ----'

[2.2]    request = """
            <session version='0.2.1' name='wwchen' type='7'
            user='wwchen' email='wwchen@nist.gov' priority='0'>
```

```

    <group name='group1'><dataSet></dataSet>
    <reduce classname='Chisq'/>
    <task cmd='echo.py'><bufsize value='3000'></task>
    <joblist name='joblist1' cnt='5'>
        <input>This is a echo example</input>
    </joblist>
    </group></session>
"""

```

```
[2.3] reply.GetOutQueue().put(request)
```

```
print '\n --- Main Thread is sleeping ----'
```

```
[2.4] time.sleep(WAITING_TIME)
```

```
except:
    print 'NetWorkThread exception: ', traceback.format_exc()
```

```
if networkThread is not None:
    print '\n --- Stop network Thread ----'
```

```
[2.5] networkThread.stop()
```

```
print '\n --- Main Thread is closing ----'
```

```
#####
```

```
if __name__ == '__main__':
```

```
    main()
```

```
#####
```

This client-side application is running in the following order:

(1) Create a new thread to handle the network, such as submit the request to the server and receive the reply from the server. This network thread has been defined in `NetworkThread` class. It can be initialized in the main thread by:

```
NetworkThread(server, reply)
```

where

Server: a tuple of `(server, port)` where `server` is the string name of the server, and `port` the number of port that the server will accept the request.

Reply: an object to handle the reply from the server. It inherits from `reply.ParkReply` class. The user needs to subclass this class, and at least implement the function `_OnReply(self, msg)`, where `msg` is the string received from the server.

The network thread will automatically call the corresponding function of `reply.ParkReply` class to handle the network message. In this example implementation, we only print out the reply message, as defined in class `replyEcho.ParkReplyEcho`:

```
#!/usr/bin/env python
```

```
#####
```

```
"""
```

```
    The class to handle the reply from the park server
```

```
"""
```

```
#####
```

```
from reply import ParkReply
```

```
#####
```

```
#####
class ParkReplyEcho(ParkReply):
    """
        This is the class to handle the network message. It is
        used by the network thread. Not used directly by the user.
    """
#####
    def __init__(self):
        super(ParkReplyEcho, self).__init__()

#####

    def _OnReply(self, msg):
        """
            Handler for the normal reply from the PARK server.
            The subclass must implement this function.
        """
        print 'Reply from parkServer:', msg

#####
```

(2) Start the network thread, prepare and submit the job request:

The network thread will be starting to handle the network by call its `start()` method [2.1]. Then the main thread needs to prepare the job request [2.2]. For the network thread use asynchronous model to communicate with the server, the job request will be stored in the queue provide by reply handler [2.3]. The user doesn't need to submit the request to the server directly. Then the main thread sleeps for a while so the network thread can receive the reply from the server, and call the reply handler to parse the reply properly [2.4]. In the real application, the main thread may in an infinite loop to prepare the job request and the reply is handled in a separated thread. Once all the work has been done, the main thread call the `stop()` method of the network thread to stop the communication with the server and exit the program.

(3) XML format for the job request [2.2]:

The job request sent to PARK server must be a valid XML string with the proper format. Among which the most import ones are highlighted in [2.2]. They are:

```
<reduce classname='Chisq' /> # classname for the reduce function
<task cmd='echo.py'>...</task> # file name for the map function
<joblist cnt='5'> # cnt is the total number of map function
    # that should run in the working nodes.
    <input>This is a echo example</input> # This is the string
    # sent to the map function.
</joblist>
```

There can have multiple `<joblist> ... </joblist>` in one job request. The reduce function must be accessible for the PARK server, and the map function must be accessible for the working nodes.

Theoretically, the map function will be run total **cnt** times with the given input on the working node, then the reduce function will be called on the server once, with the output from **cnt** map functions as the input. For the map functions are not running simultaneously in distributed environment, the reduce function will be called on the server each time when there is the reply from the working node.

2) Server-Side Application (Map function)

This kind of application will run in the working nodes. The PARK working nodes run this map function as a separated thread or process. All the map function outputs to the standard out will be collected by the PARK working server, and send it back to the PARK service server. The PARK service server stores this information in the archive system, passes it to the reduce function, and sends the results to the client.

The map functions (services) are under `~/park/services` directories. Echo service (`~/park/services/svrcommon/echo.py`) is a simple service which only echoes the string passed to it. The function is:

```
#!/usr/bin/env python

#####
import os, sys
import traceback
#####

#####
""" The Echo service that send back the user's xml request """
from commonsvrUtil import service, serviceFile
#####

#####
def echo(s0, sn = 0):
[0]    print s0
#####

#####
FILENAME = os.path.join(os.getcwd(), 'echo.xml')
ECHO_SERVICE = 'Echo service'
#####
#####
if __name__ == '__main__':
    #serviceFile(echo, FILENAME, ECHO_SERVICE)
    service(echo, ECHO_SERVICE)
#####
```

A utility function `service` is defined in `commonsvrUtil.py`, which will handle the communication with the PARK working server. It reads the input from the standard input, and calls the given service. The corresponding `serviceFile` will read the input from a file, and is normally used for the debugging. The first parameter for these two utility functions is a service function, and the last one is the description string about the service. The service function has the following format:

```
Function_name(s0, sn = 0)
```

Where

`s0`: an xml string that specified in the `<input>...</input>` tag in the job request. The service function needs to parse this xml string to get the input parameters.

`sn`: A integer representing the serial number of current job within the `cnt` number of requested jobs.

Within the service function, any outputs to the standard output will be collected and sent back to the client. In echo service, it will send back the input string, as shown in [0].

Recommendations for distributing service:

1. Put the service under correct directory, currently put it on any subdirectories under `~/park/services`.
2. Make sure that the service works locally, for example using `serviceFile` to make sure the service works correctly, then changing to `service` for the service application.
3. Make sure that the service is executable in Unix/Linux environment.